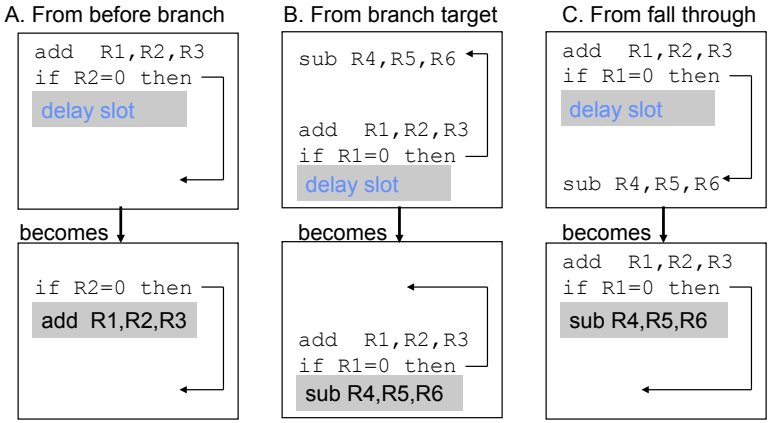
CMSC 411
Computer Systems Architecture
Lecture 6
Basic Pipelining (cont.)

Alan Sussman
als@cs.umd.edu

Administrivia

- HW#1 due today
- Answers to HW #1 posted soon
  - password protected, with instructions sent via email
- Homework problems for Unit 3 posted soon
- First project, on pipelining, out soon too

Scheduling Branch Delay Slots



- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

Scheduling branch delay slot

- If taken from before branch
  - branch must not depend on rescheduled instruction
  - always improves performance
- If taken from branch target
  - must be OK to execute rescheduled instructions if branch not taken, and may need to duplicate insts.
  - performance improved when branch taken
- If taken from fall through
  - must be OK to execute insts. if branch taken
  - improves performance when branch not taken

Delayed Branch

- **Compiler effectiveness for single branch delay slot:**
  - **Fills about 60% of branch delay slots**
  - **About 80% of instructions executed in branch delay slots useful in computation**
  - **About 50% (60% x 80%) of slots usefully filled**
- **Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot**
  - **Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches**
  - **Growth in available transistors has made dynamic approaches relatively cheaper**

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume: **4% unconditional branch,**
6% conditional branch- untaken,
10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict taken	1	1.20	4.2	1.33
Predict not taken	1	1.14	4.4	1.40
Delayed branch	0.5	1.10	4.5	1.45

Pipelining Summary

- **Pipelining can speed instruction execution (throughput)**
- **But need to deal with structural hazards, data hazards, and control hazards**
- **Next**
  - **How to handle exceptions?**
  - **How to handle long instructions, such as floating point arithmetic?**

The problem

- **Question: What makes pipelining hard to implement?**
- Answer: **Surprises**
- **Technical names for surprises:**
  - **exceptions**
  - **faults**
  - **interrupts**

Some examples of exceptions

- **Request for I/O**
- **Arithmetic troubles: overflow or underflow**
- **Page fault: data not in (physical) memory**
- **Illegal address, giving a memory protection violation**
- **Hardware failure**

Classifying exceptions

- Synchronous**: repeatable every time**
  - **Example: DIV R2, R2, R0**

  Asynchronous**: caused by external events like hardware failure and devices external to processor and memory**
- User requested**: user task asks for it (example: breakpoint)**

  Coerced**: cannot be predicted by user**
- User maskable**: can be disabled by user task**
  - **Example: arithmetic exception**

  Nonmaskable**: cannot be turned off**
  - **Example: hardware failure**

Classifying exceptions (cont.)

- Within instruction**: prevents instruction from completing**
  Between instructions**: no instruction prevented**
- Terminating**: stops the task**
  Resuming**: task can continue**
- **Machines that handle exceptions, save the state, and then restart correctly are said to be** restartable

Categorizing exceptions

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynch	Coerced	Not	Between	Resume
Invoke OS	Synch	User req.	Not	Between	Resume
Tracing instructions	Synch	User req.	Maskable	Between	Resume
Breakpoint	Synch	User req.	Maskable	Between	Resume
Integer overflow	Synch	Coerced	Maskable	Within	Resume
Floating pt. overflow/ underflow	Synch	Coerced	Maskable	Within	Resume

Categorizing exceptions (cont.)

Exception type	Synch. vs. asynch.	User request vs. coerce	User maskable vs. not	Within vs. between instructions	Resume vs. terminate
Page fault	Synch	Coerced	Not	Within	Resume
Misaligned memory access	Synch	Coerced	Maskable	Within	Resume
Mem. prot. violation	Synch	Coerced	Not	Within	Resume
Undefined instruction	Synch	Coerced	Not	Within	Terminate
Hardware malfunction	Asynch	Coerced	Not	Within	Terminate
Power failure	Asynch	Coerced	Not	Within	Terminate

The most difficult exceptions...

- **... are those that occur within EX or MEM stages and need to be handled in a restartable way**
- **Why difficult? Handling one includes:**
  - **the next IF gets a "trap instruction"**
  - **until the trap is taken, turn off all "writes" for the faulting instruction and those that follow it**
  - **what does the trap do?**
    - » **The trap transfers control to the exception handling routine in the operating system, which saves the PC of the faulting instruction and handles the fault**
  - **the task is then resumed, using the saved PC and the MIPS instruction RFE or something like it**
- Note**: May need to save several PCs if delayed branches are involved**

Exceptions (cont.)

- **Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction -** precise exception handling
- **This is the right way to do it, but sometimes architects/manufacturers take shortcuts**

Review: Exceptions

- **Ideally, pipeline can be interrupted so that instructions before the fault complete. Then want to restart execution just after the faulting instruction -** precise exception handling

When do MIPS exceptions occur?

- IF
  - page fault on instruction fetch
  - misaligned memory access
  - memory protection violation
- ID
  - undefined or illegal opcode
- EX
  - arithmetic exception
- MEM
  - page fault on data fetch/store
  - misaligned memory access
  - memory protection violation
- WB: None!

Examples of exception handling

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- Handle the MEM fault first, then restart

LD	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

- IF fault occurs first, even though LD will fault later
- But for precise exceptions, must handle LD fault first

How is this done?

- Answer: **Don't handle exceptions until the WB stage**
  - each instruction has an associated status vector that keeps track of faults
  - any bit set in the status vector turns off register writes and memory writes
  - in WB stage, the status vector is checked and any fault is handled
  - So, since instructions reach WB in proper order, faults for earlier instructions are handled *before* faults for later instructions
    - » Unfortunately, will need to violate this later (for instructions that don't reach WB in proper order)

Commitment

- When an instruction is guaranteed to complete, it is committed
- Life is easier if no instruction changes the permanent machine state before it is committed
- In MIPS, commitment occurs at the end of the MEM stage - that's why register update occurs in the stage after that
- Some machines muddy the state before commitment, and the exception handler must do its best to restore the state that existed before the instruction started

Complications with long instructions

- So far, all MIPS instructions take 5 cycles
- But haven't talked yet about the floating point instructions
- Take it on faith that floating point instructions are inherently slower than integer arithmetic instructions
  - doubters may consult Appendix I in H&P online

How slow is slow?

- Some typical times:
  - latency **is the number of cycles between an instruction that produces a result and one that uses it**
  - initiation interval **is the number of cycles between two instructions of the same kind (for example, two ADD.Fs)**

Instruction	Latency	Initiation
ALU uses	0	1
Load/store	1	1
ADD.F,SUB.F	3	1
DIV.F	24	25